
# Object Oriented Programming with Java 8

**Dr.Akshita Chanchlani**

# Contents

- Java Buzzwords (Features)
- Access Specifier
- Class and Object
- Instantiation
- Reference
- this
- Constructor and its types
- Object class
- toString()
- Static keyword
- Static field
- Static member function
- Static initializer block
- Static import

# Java Buzzwords

1. Simple
2. Object Oriented
3. Architecture Neutral
4. Portable
5. Robust
6. Multithreaded
7. Dynamic
8. Secure
9. High Performance
10. Distributed

# Simple

- Java is **simple** programming language.
  - **Syntax of Java is simpler than syntax of C/C++ hence it is considered as simple**

    - Ø No need of header files and macros.
    - Ø We can not define anything global
    - Ø Do not support structure and union. operator overloading.
    - Ø Do not support copy constructor and assignment operator function constructor member initializer list and default argument constant data member and constant member function.
      Delete operator, destructor , friend function, friend class.
      Multiple class (Multiple inheritance)

    - Ø `No diamond problem and virtual base class.`

    - Ø `Do not support pointer and pointer arithmetic.`

  - **Size of software(JDK), that is required to develop Java application is small hence Java is considered as simple programming language.**

- Java does not support multiple inheritance. To some extent, the interface feature provides the desirable features of multiple inheritance to a Java program without some of the underlying problems.(death of a diamond)

# Object Oriented

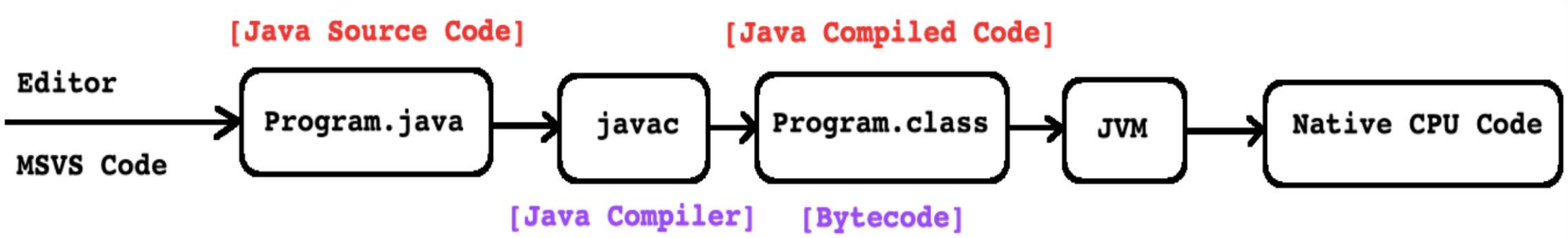
- Java is **object oriented** programming language.
  - Java Supports all the major and minor pillars of oops hence it is considered as object oriented programming language.

  - **Major pillars of oops.**
    1. Abstraction
    2. Encapsulation
    3. Modularity
    4. Hierarchy

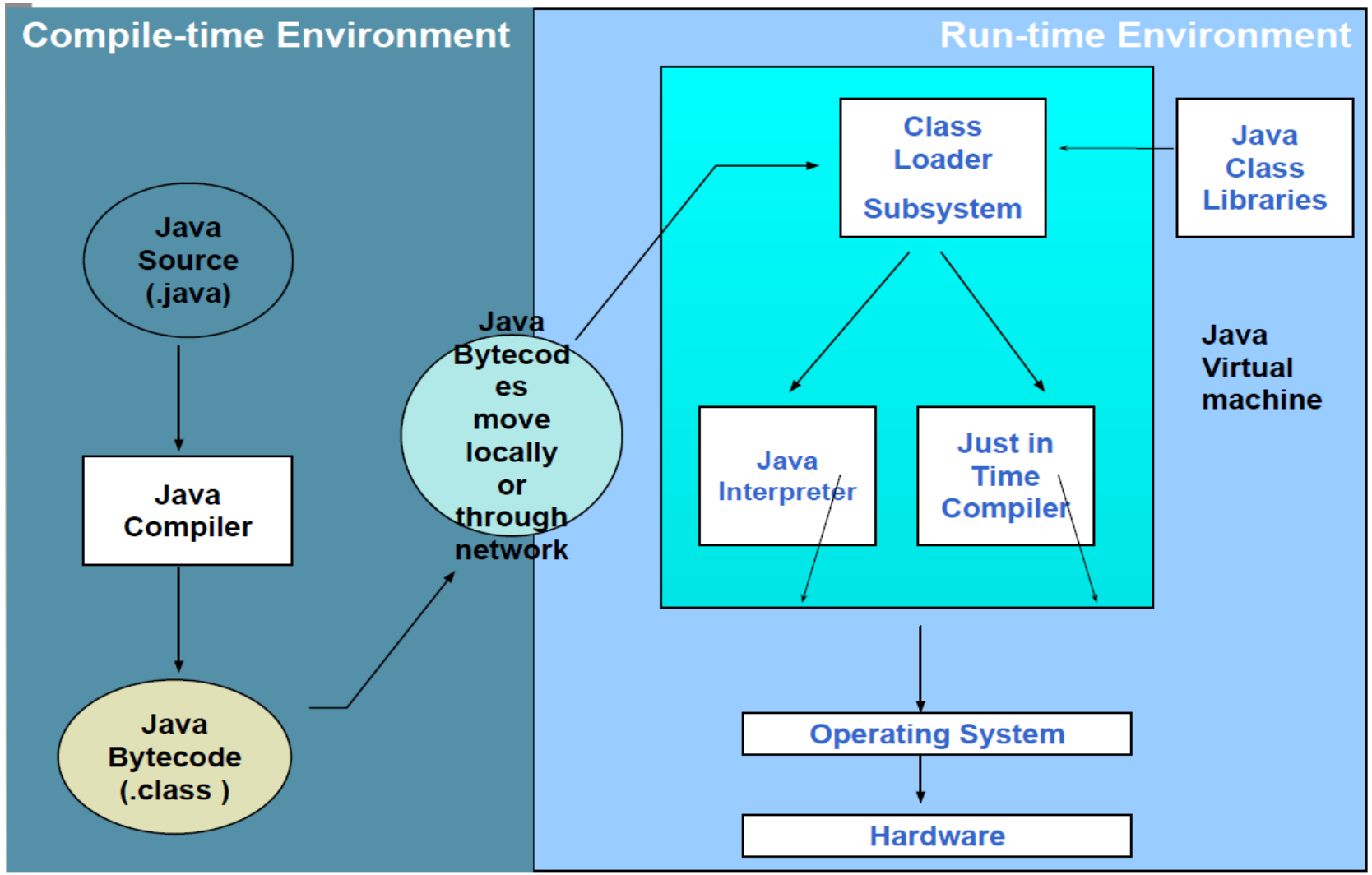
  - **Minor pillars of oops.**
    1. Typing / Polymorphism
    2. Concurrency
    3. Persistence.

# Architecture Neutral

- Java is object **architecture neutral** programming language.
  - Java technology is designed to support applications that will be deployed into heterogeneous network environments. In such environments, applications must be capable of executing on a variety of hardware architectures. Within this variety of hardware platforms, applications must execute on the top of a variety of operating systems. To accommodate the diversity of operating environments, the Java Compilerproduct generates *bytecodes*--an *architecture neutral* intermediate format designed to transport code efficiently to multiple hardware and software platforms.



# How it Works

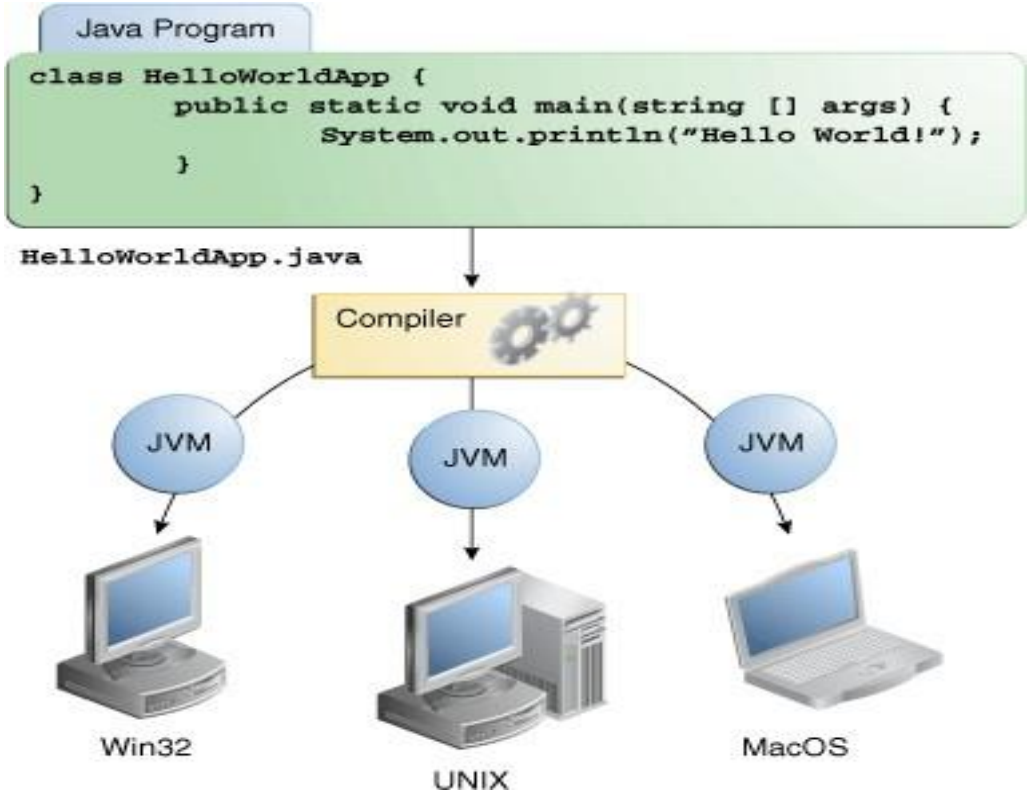

Compile-time Environment
Run-time Environment
Java Source (.java)
Java Compiler
Java Bytecode (.class )
Java Bytecodes move locally or through network
Class Loader Subsystem
Java Class Libraries
Java Virtual machine
Java Interpreter
Just in Time Compiler
Operating System
Hardware

# Java Platform Independent

Java is independent only for one reason:
– Only depends on the Java Virtual Machine (JVM),
– code is compiled to bytecode, which is interpreted by the resident JVM,
– JIT (just in time) compilers attempt to increase speed.

# Portable

- Java is **portable** programming language.
  - Architecture neutrality is just one part of a truly *portable* system .



# Portable

- Java is **portable** programming language.
  - Java technology takes portability a stage further by being strict in its definition of the basic language.
  - Java technology puts a stake in the ground and specifies the sizes of its basic types and the behavior of its data arithmetic operators.
  - Your programs are the same on every platform--there are no data type incompatibilities across hardware and software architectures.

| Sr.No. | Primitive Type | Size | Default Value For Field |
|---|---|---|---|
| 1 | boolean | Isn't Defined | FALSE |
| 2 | byte | 1 Byte | 0 |
| 3 | char | 2 Bytes | \u0000' |
| 4 | short | 2 Bytes | 0 |
| 5 | int | 4 Bytes | 0 |
| 6 | float | 4 Bytes | 0.0f |
| 7 | double | 8 Bytes | 0.0d |
| 8 | long | 8 Bytes | 0L |

# Robust

- Java is **robust** programming language.
  - The Java programming language is designed for creating highly *reliable* software. It provides extensive compile-time checking, followed by a second level of run- time checking.
  - Java is robust because of following features:

1. *Architecture Neutral*.
   - Ø Java developer is free from developing H/W or OS specific coding.
2. *Object orientation*.
   - Ø Reusability reduces developer's effort.
3. *Automatic memory management*.
   - Ø Developer need not to worry about memory leakage / program crashes.
4. *Exception handling*.
   - Ø Java compiler helps developer to provide try-catch block.

# Multithreaded

- Java is **multithreaded** programming language.
  - When we start execution of Java application then JVM starts execution threads hence Java is considered as multithreaded.
    1. Main thread
       - Ø It is user thread / non daemon thread.
       - Ø It is responsible for invoking main method.
       - Ø Its default priority is 5( Thread.NORM_PRIORITY ).
    2. Garbage Collector / Finalizer
       - Ø It is daemon thread / background thread.
       - Ø It is responsible for releasing / deallocating memory of unused objects.
       - Ø Its default priority is 8( Thread.NORM_PRIORITY + 3 ).
  - The Java platform supports multithreading at the language level with the addition of sophisticated synchronization primitives: the language library provides the Thread class, and the run-time system provides monitor and condition lock primitives. At the library level, moreover, Java technology's high-level system libraries have been written to be thread safe: the functionality provided by the libraries is available without conflict to multiple concurrent threads of execution.

# Dynamic

- Java is **dynamic** programming language.
  - While the Java Compiler is strict in its compile-time static checking, the language and run-time system are *dynamic* in their linking stages. Classes are linked only as needed. New code modules can be linked in on demand from a variety of sources, even from sources across a network.
  - Java is designed to adapt to an evolving environment.
  - Libraries can freely add new methods and instance variables without any effect on their clients.
  - In Java finding out runtime type information is straightforward.
  - In Java, all the methods are by default virtual.

# Secure

- Java is **secure** programming language.
  - Java is intended to be used in networked/distributed environments. Toward that end, a lot of emphasis has been placed on security. Java enables the construction of virus-free, tamper-free systems.
  - From the beginning, Java was designed to make certain kinds of attacks impossible, among them:
    1. Overrunning the runtime stack—a common attack of worms and viruses
    2. Corrupting memory outside its own process space
    3. Reading or writing files without permission

Pointer denial - reduces chances of virulent(harmful – virus prone)
programs corrupting host,
■ Applets even more restricted -
– May not
• run local executables,
• Read or write to local file system,
• Communicate with any server other than the originating server.

# High Performance

- Java

  is **high performance** programming language.

  - The Java platform achieves superior performance by adopting a scheme by which the interpreter can run at full speed without needing to check the run-time environment.
  - The *automatic garbage collector* runs as a low-priority background thread, ensuring a high probability that memory is available when required, leading to better performance.
  - Applications requiring large amounts of compute power can be designed such that compute-intensive sections can be rewritten in native machine code as required and interfaced with the Java platform.
  - In general, users perceive that interactive applications respond quickly even though they're interpreted.

# Distributed

- Java is **distributed** programming language.
- Java has an extensive library of routines for coping with protocols like HTTP , TCP/IP and FTP.
- Java applications can open and access objects across the Net via URL with the same ease as when accessing a local file system.

o Nowadays, one takes this for granted, but in 1995, connecting to a web server from a C++ or Visual Basic program was a major undertaking.

# How to access members of package?

Package : p1

```
public class Complex{
  //TODO
}
```

```
public class Program{
  public static void main( String[] args ){
    p1.Complex c1 = new p1.Complex( );
  }
}
```

1

```
import p1.Complex;
public class Program{
  public static void main( String[] args ){
    Complex c1 = new Complex( );
  }
}
```

2

# Modifier

1. ABSTRACT
2. FINAL
3. INTERFACE
4. NATIVE
5. PRIVATE
6. PROTECTED
7. PUBLIC
8. STATIC
9. STRICT
10. SYNCHRONIZED
11. TRANSIENT
12. VOLATILE

# Access Modifier

- If we want to control visibility of members of class then we should use access modifier.
- There are 4 access modifiers in Java:
    1. private
    2. package-level private / default
    3. protected
    4. public

| | Same Package | | | Different Package | |
|---|---|---|---|---|---|
| Access Modifiers | Same class | Sub class | Non sub cass | Sub class | Non Sub class |
| private | A | NA | NA | NA | NA |
| package level private/Default | A | A | A | NA | NA |
| protected | A | A | A | A | NA |
| public | A | A | A | A | A |

# Class

- Consider following examples:
  1. day, month, year - related to – Date
  2. hour, minute, second - related to – Time
  3. red, green, blue - related to Color
  4. real, imag - related to – Complex
  5. xPosition, yPosition - related to Point
  6. number, type, balance - related to Account
  7. name, id, salary - related to Employee

- If we want to group related data elements together then we should use/define class in Java.

```
class Date{
    int day;        //Field
    int month;      //Field
    int year;       //Field
}
```

```
class Employee{
    String name;    //Field
    int id;         //Field
    float salary;   //Field
}
```

# Class and object

- class is a non primitive/reference type in Java.
- Classes and Objects are basic concepts of Object Oriented Programming which revolve around the real life entities.

A **class** is a user defined blueprint or prototype or template : from which objects are created.
It represents the set of properties(DATA) and methods(ACTIONs) that are common to all objects of one type.

**Class declaration includes**

1. Access specifiers : A class can be public or has default access
2. Class name: The name should begin with a capital letter & then follow camel case convention
3. Superclass(if any): The name of the class's parent (superclass), if any, preceded by the keyword extends. (Implicit super class of all java classes is java.lang.Object)
4. Interfaces(if any): A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
   eg : public class Emp extends Person implements Runnable,Comparable{...}
5. Body: The class body surrounded by braces, { }.
6. Constructors are used for initializing state of the new object/s.
7. Fields are variables that provides the state of the class and its objects
8. Methods are used to implement the behavior of the class and its objects.

eg : Student,Employee,Flight,PurchaseOrder, Shape ,BankAccount.....

# Object

- It is a basic unit of Object Oriented Programming and represents the real life entities. A typical Java program creates many objects, which interact by invoking methods.

- An object consists of :

  - State : It is represented by attributes of an object. (properties of an object) / instance variables(non static)
  - Behavior : It is represented by methods of an object (actions upon data)
  - Identity : It gives a unique identity to an object and enables one object to interact with other objects. eg : Emp id / Student PRN / Invoice No

- Creating an object
  - The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The new operator also invokes the class constructor.

# Class

- Field
  - Ø A variable declared inside class / class scope is called a field.
  - Ø Field is also called as attribute or property.
- Method
  - Ø A function implemented inside class/class scope is called as method.
  - Ø Method is also called as operation, behavior or message.
- Class
  - Ø Class is a collection of fields and methods.
  - Ø Class can contain
    1. Nested Type
    2. Field
    3. Constructor
    4. Method
- **Instance** : In Java, Object is also called as instance.

Note: All stand-alone C++ programs require a function named main and can have numerous other functions. Java does not have stand alone functions, all functions (called methods) are members of a class. All classes in Java ultimately inherit from the Object class, while it is possible to create inheritance trees that are completely unrelated to one another in C++. In this sense , Java is a pure Object oriented language, while C++ is a mixture of Object oriented and structure language.

# Instantiation

- Process of creating instance/object from a class is called as instantiation.

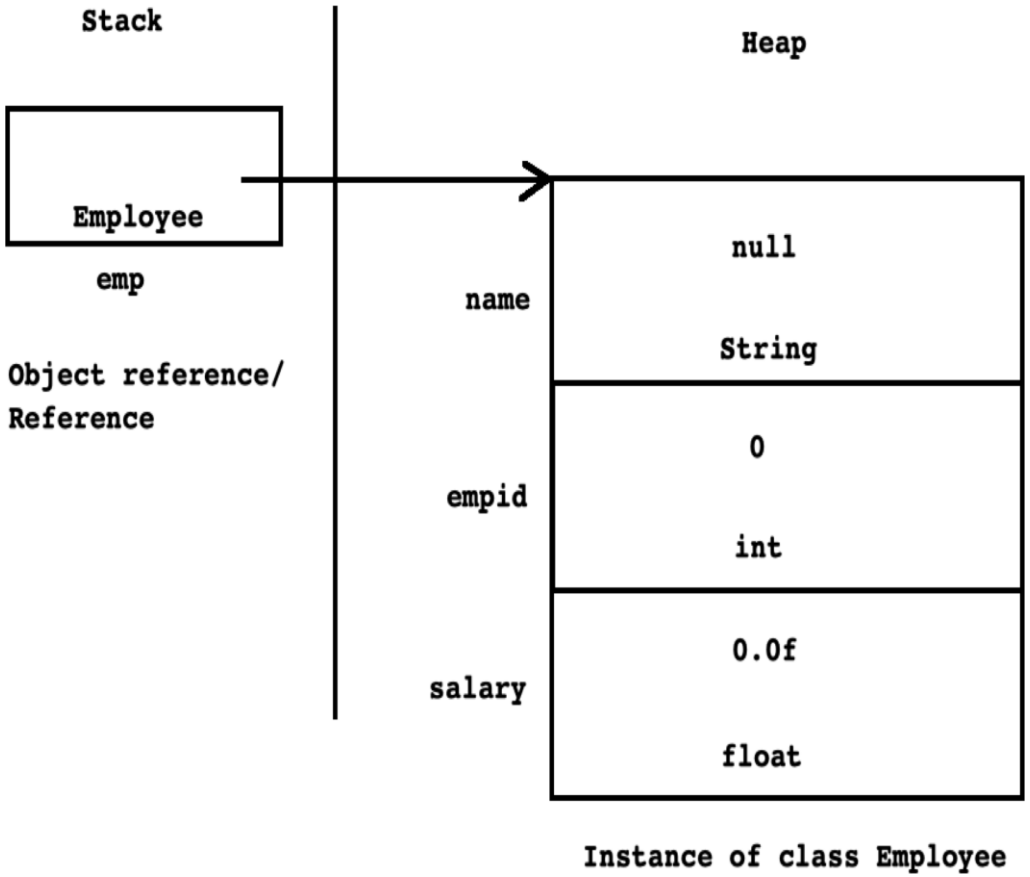
- In C programming language
  - Ø Syntax : struct StructureName identifier_name; struct
  - Ø Employee emp;

- In C++ programming language
  - Ø Syntax : [class] ClassName identifier_name;
  - Ø Employee emp;

- In Java programming language
  - Ø Syntax : ClassName identifier_name = new ClassName( );
  - Ø Employee emp = new Employee();

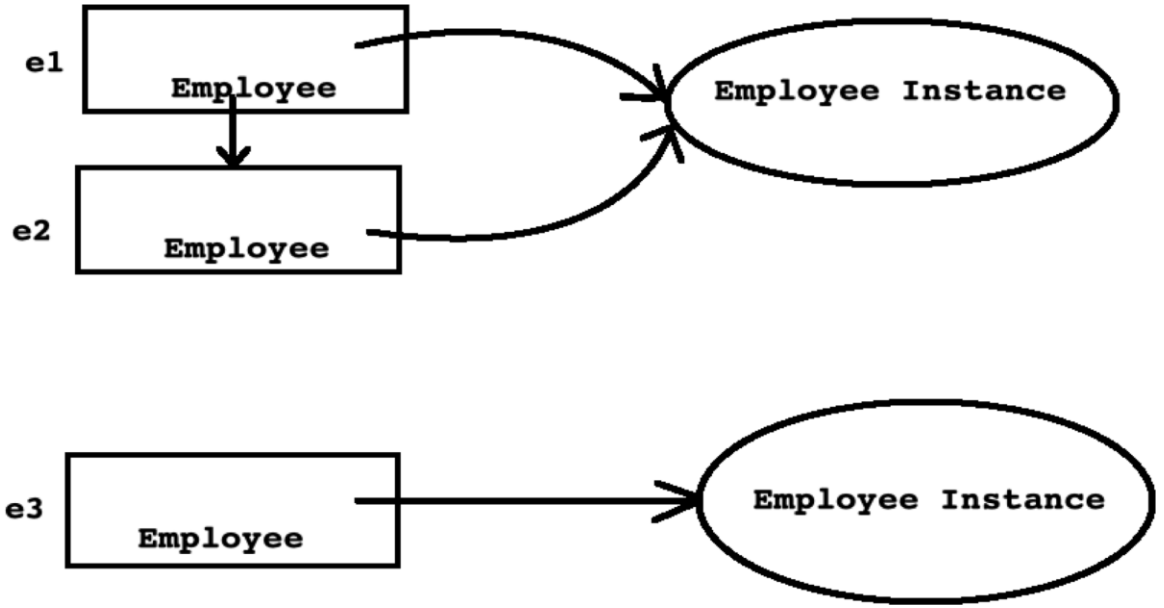
- **Every instance on heap section is anonymous.**

# Instantiation

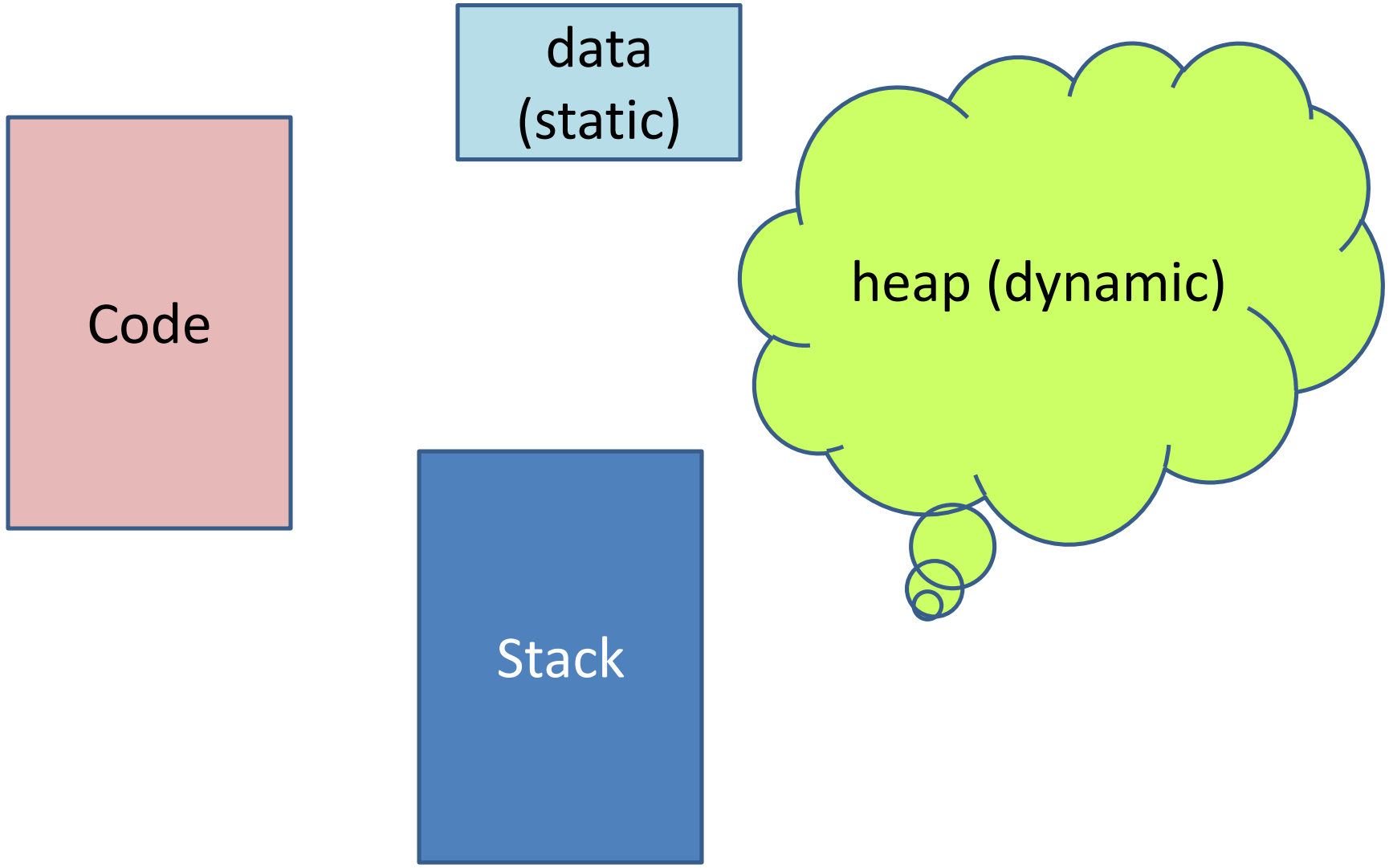



For eg :

```
1. Employee e1 = new Employee();
2. Employee e2 = e1;
3. Employee e3 = new Employee();
```



# Types of memory used by executable task



# this current object reference

- If we call non static method on instance( actually object reference ) then compiler implicitly pass, reference of current/calling instance as a argument to the method implicitly. To store reference of current/calling instance, compiler implicitly declare one reference as a parameter inside method. It is called this reference.
- **Using this reference, non static fields and non static methods are communicating with each other. Hence this reference is considered as a link/connection between them.**
- Definition
  - Ø **“this" is implicit reference variable that is available in every non static method of class which is used to store reference of current/calling instance.**
- Inside method, to access members of same class, use this keyword is optional

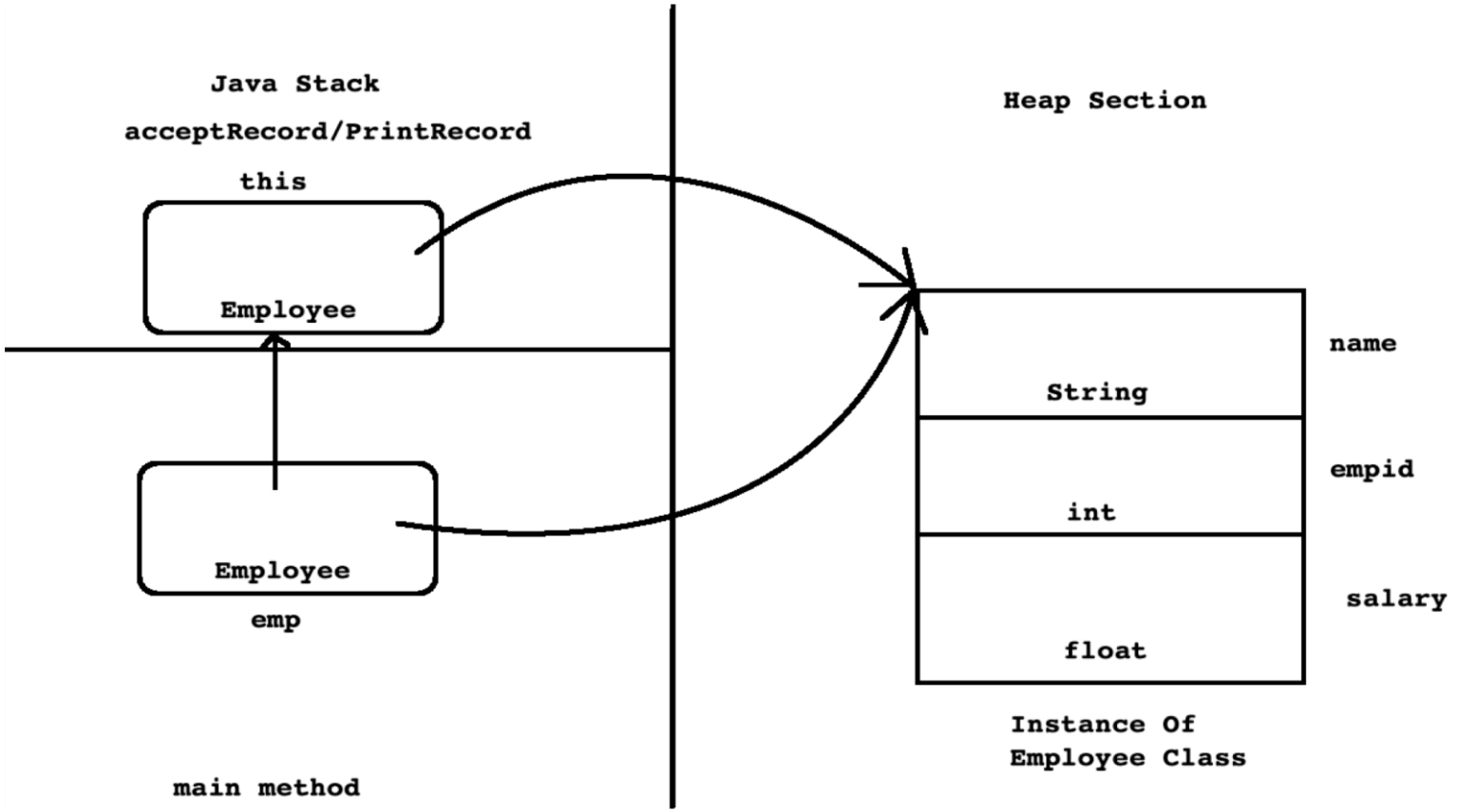
**Uses of this keyword :**

1. To unhide , instance variables from method local variables.(to resolve the conflict)

eg : this.name=name;

2. To invoke the constructor , from another overloaded constructor in the same class.(constructor chaining , to avoid duplication)

# this reference



# this reference

- If name of local variable/parameter and name of field is same then preference is always given to the local variable.

```
class Employee{
    private String name;
    private int empid;
    private float salary;
    public void initEmployee(String name, int empid, float salary ){
        this.name = name;
        this.empid = empid;
        this.salary = salary;
    }
}
```

# Constructor

- If we want to initialize instance then we should define constructor inside class.
- Constructor look like method but it is not considered as method.
- It is special because:
  - Its name is same as class name.
  - It doesn't have any return type.
  - It is designed to be called implicitly
  - It is called once per instance.
- We can not call constructor on instance explicitly

  **Employee emp = new Employee();**
  **emp.Employee( );//Not Ok**

- **Types of constructor:**
  1. Parameterless constructor
  2. Parameterized constructor
  3. Default constructor .

# Parameterless Constructor

- If we define constructor without parameter then it is called as parameterless constructor.
- It is also called as zero argument / user defined default constructor.
- If we create instance without passing argument then parameterless constructor gets called.

```
public Employee( ){
    //TODO
}
```

```
Employee emp = new Employee( ); //Here on instance parameterless ctor will call.
```

# Parameterized Constructor

- If we define constructor with parameter then it is called as parameterized constructor.
- If we create instance by passing argument then parameterized constructor gets called.

```
public Employee( String name, int empid, float salary ){
    //TODO
}
```

```
Employee emp = new Employee( "ABC",123, 8000 ); //Here on instance parameterized ctor will call.
```

# Default Constructor

- If we do not define any constructor inside class then compiler generate one constructor for the class by default. It is called default constructor.

- Compiler generated default constructor is parameterless.
- Compiler never generate default parameterized constructor. In other words, if we want to create instance by passing arguments then we must define parameterized constructor inside class.

# Constructor Chaining

- We can call constructor from another constructor. It is called constructor chaining.
- For constructor chaining, we should use this statement.
- this statement must be first statement inside constructor body.
- Using constructor chaining, we can reduce developers effort.

```
class Employee{
    //TODO : Field declaration
    public Employee( ){
        this( "None", 0, 8500 );    //Constructor Chaining
    }
    public Employee( String name, int empid, float salary ){
        this.name = name;
        this.empid = empid;
        this.salary = salary;
    }
}
```

# Why java doesn't support c++ copy constructor?

Java does. They're just not called implicitly like they are in C++ .

Firstly, a copy constructor is nothing more than:

```
public class Demo {
  private int val;

  public Demo() { } // public no-args constructor
  public Demo(Demo b) { val = b.val; }  // copy constructor
}
```

Now C++ will implicitly call the copy constructor with a statement like this:

```
Demo b2 = b1;
```

Cloning/copying in that instance simply makes no sense in Java because all b1 and b2 are references and not value objects like they are in C++. In C++ that statement makes a copy of the object's state. In Java it simply copies the reference. The object's state is not copied so implicitly calling the copy constructor makes no sense.

# Literals

- Consider following literals in Java:
  1. true : boolean
  2. 'A' : char ch;
  3. “Akshita" : String str;
  4. 123 : int num1;
  5. 72.93f : float num2
  6. 3.142 : double num3
  7. null : Used to initialize reference variable.

- null is a literal which is designed to initialize reference variable
  - int num = null ; //invalid
  - Integer num=null; // VALID
  - String str=null; // VALID
  - Employee emp=null; // VALID

# null

If reference variable contains null value then it is called null reference  variable / null object.

```
class Program{
    public static void main(String[] args) {
      Employee emp; //Object reference / reference
      emp.printRecord(); //error: variable emp might not have been initialized
    }
}
```

```
public static void main(String[] args) {
    Employee emp = null; //null reference varibale / null object
    emp.printRecord();    //NullPointerException
}
```

# Value type VS Reference Type

| Sr. No. | Value Type | Reference Type |
|---|---|---|
| 1 | primitive type is also calle as value type. | Non primitive type is also called as reference type. |
| 2 | boolean, byte, char, short, int, float, double, long are primitive/value type. | Interface, class, type variable and array are non primitive/reference type. |
| 3 | Variable of value type contains value. | Variable of reference type contains reference. |
| 4 | Variable of value type by default contains 0 value. | Variable of reference type by default contain null reference. |
| 5 | We can not create variable of value type using new operator. | It is madatory to use new operator to create instance of reference type. |
| 6 | variable of value type get space on Java stack. | Intstance of reference type get space on heap section. |
| 7 | We can not store null value inside variable of value type. | We can store null value inside variable reference type. |
| 8 | In case of copy, value gets copied. | In case of copy, reference gets copied. |

# Coding Convention

- **Pascal Case Coding/Naming Convention:**
  - Example
    1. System
    2. StringBuilder
    3. NullPointerException
    4. IndexOutOfBoundsException

  - In this case, including first word, first character of each word must in upper case.

  - We should use this convention for:
    1. Type Name( Interface, class, Enum, Annotation )
    2. File Name

# Coding Convention

- **Camel Case Coding/Naming Convention:**
  - Example
    1. main
    2. parseInt
    3. showInputDialog
    4. addNumberOfDays
  - In this case, excluding first word, first character of each word must in upper case.
  - We should use this convention for:
    1. Method Parameter and Local variable
    2. Field
    3. Method
    4. Reference

# Coding Convention

- **Naming Convention for package:**
  - We can specify name of the package in uppercase as well as lower-case. But generally it is mentioned in lower case.
  - Example
    1. java.lang
    2. java.lang.reflect
    3. java.util
    4. java.io
    5. java.net
    6. java.sql

# Coding Convention

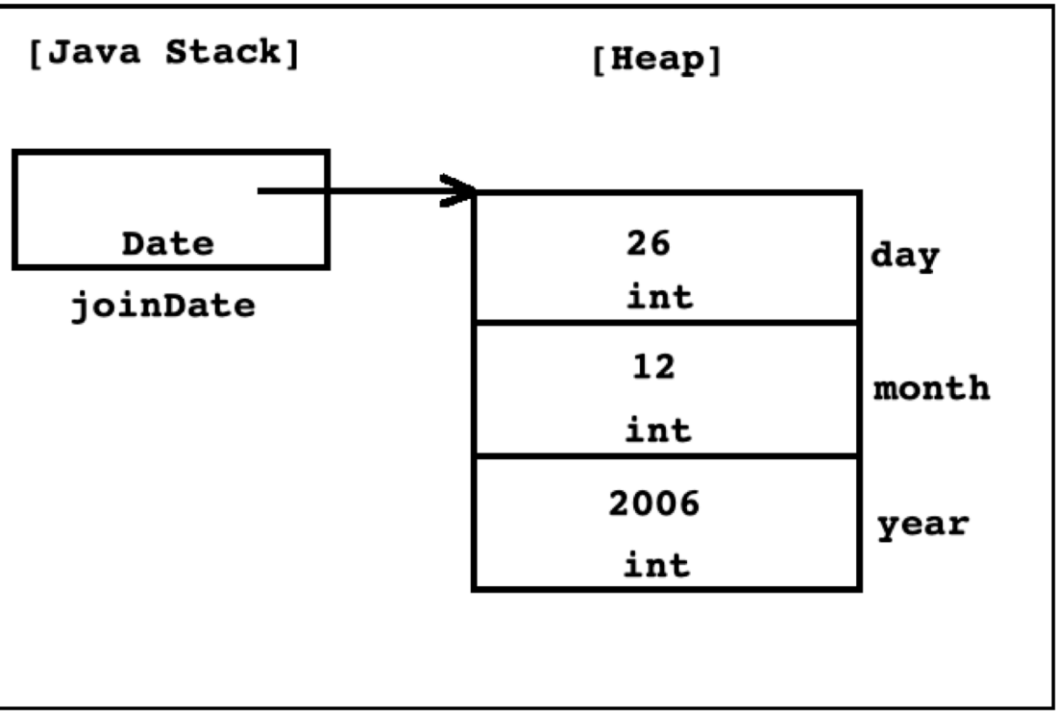
- **Naming Convention for constant variable and enum constant:**
  - Example

```
1. public static final int SIZE;
2. enum Color{ RED, GREEN, BLUE }
3. Name of the final variable and name of the enum constant should be in upper case.
```

# Reference

- Local reference variable get space on Java Stack.

```
class Date{
    private int day;
    private int month;
    private int year;
}
class Program{
    public static void main(String[] args) {
        Date joinDate = new Date( 26,12,2006);
        //joinDate : Method Local Variable
    }
}
```




- In above code joinDate is method local reference variable hence it gets space on Java Stack.

# Reference

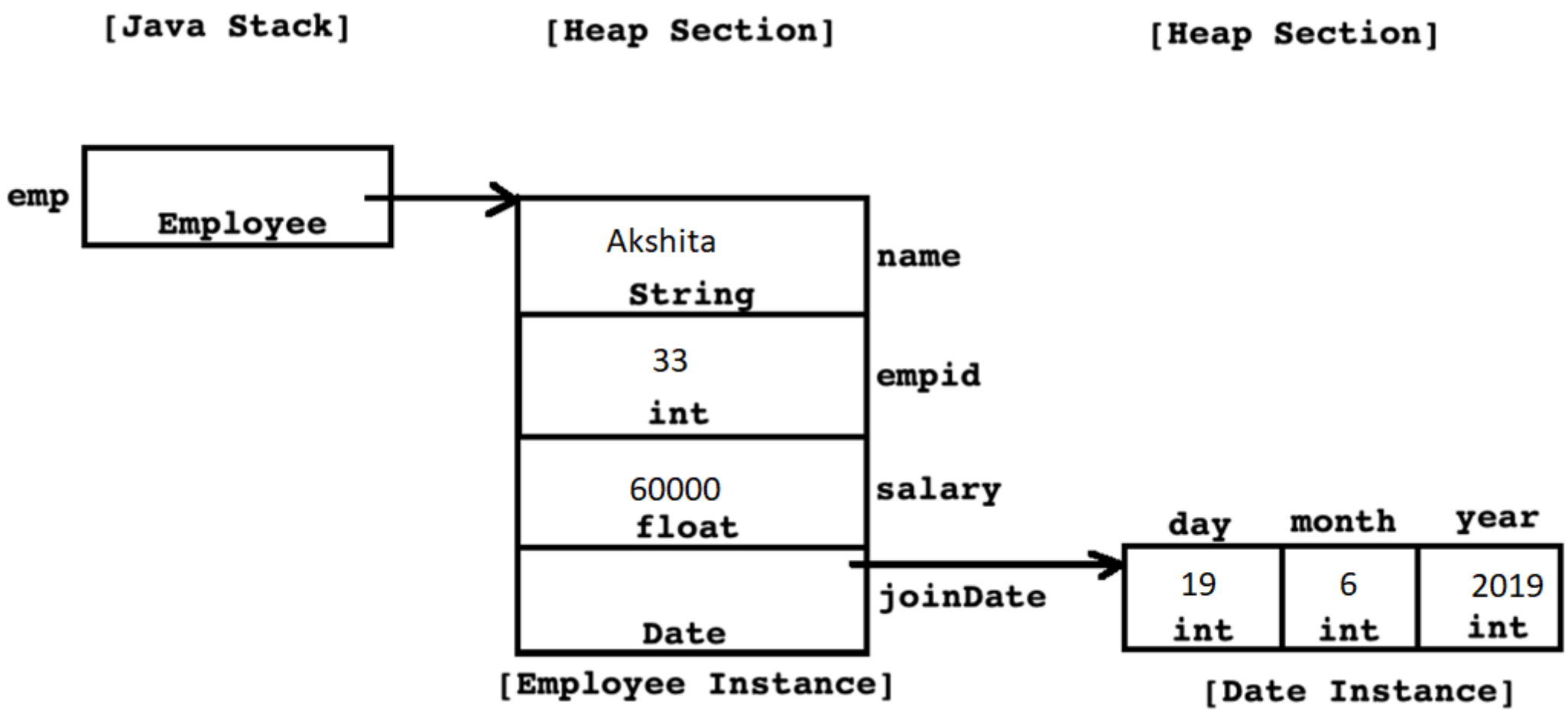
- Class scope reference variable get space on heap.

```
class Employee{
    private String name;
    private int empid;
    private float salary
    private Date joinDate;  //joinDate : Field
    public Employee( String name, int empid, float salary, Date joinDate ){
        this.name = name;
        this.empid = empid;
        this.salary = salary;
        this.joinDate = joinDate;
    }
}
```

Class Program
{
public static void main(String[] args)
{
Employee emp = new Employee (“Akshita”,33,60000,new Date(19,06,2019));
}
}

- In above code, emp is method local reference variable hence it gets space on Java Stack. But joinDate is field of Employee class hence it will get space inside instance on Heap.

# Reference


[Java Stack]
[Heap Section]
[Heap Section]
emp
Employee
Akshita
String
name
33
int
empid
60000
float
salary
Date
joinDate
day
month
year
19
int
6
int
2019
int
[Employee Instance]
[Date Instance]

# Object class

- It is a non final and concrete class declared in java.lang package.
- In java all the classes( not interfaces )are directly or indirectly extended from java.lang.Object class.
- In other words, java.lang.Object class is ultimate base class/super cosmic base class/root of Java class hierarchy.
- Object class do not extend any class or implement any interface.
- It doesn't contain nested type as well as field.
- It contains default constructor.
  - Object o = new Object("Hello"); //Not OK
  - Object o = new Object( ); //OK

- Object class contains 11 methods.

# Object class

- Consider the following code:

```
class Person{
}
class Employee extends Person{
}
```

- In above code, java.lang.object is direct super class of class Person.

- In case class Employee, class Person is direct super class and class object is indirect super class.

# Methods Of Object class

```
1. public String toString();
2. public boolean equals(Object obj);
3. public native int hashCode( );
4. protected native Object clone( )throws CloneNotSupportedException
5. protected void finalize( void )throws Throwable

6. public final native Class<?> getClass( );
7. public final void wait( )throws InterruptedException
8. public final native void wait( long timeout)throws InterruptedException
9.public  final void wait( long timeout, int nanos)throws InterruptedException
10.public final native void notify( );
11.public final  native void notifyAll( );
```

# toString( ) method

- It is a non final method of java.lang.Object class.
- Syntax:
  - **public String toString( );**
- If we want to return state of Java instance in String form then we should use toString() method.
- Consider definition of toString inside Object class:

```
public String toString() {
    return this.getClass().getName() + "@" + Integer.toHexString(this.hashCode());
}
```

# toString( ) method

- If we do not define toString() method inside class then super class's toString() method gets call.
- If we do not define toString() method inside any class then object class's toString() method gets call.
- It return String in following form:
  - **F.Q.ClassName@HashCode**
  - **Example : test.Employee@6d06d69c**
- If we want state of instance then we should override toString() method inside class.
- The result in toString method should be a concise but informative that is easy for a person to read.
- It is recommended that all subclasses override this method.

# Final variable

- In java we do not get const keyword. But we can use final keyword.
- After storing value, if we don't want to modify it then we should declare variable final.

```
public static void main(String[] args) {
    final int number = 10;  //Intialization
    //number = number + 5;    //Not OK
    System.out.println("Number  :   "+number );
}
```

```
public static void main(String[] args) {
    final int number;
    number = 10;    //Assignment
    //number = number + 5;    //Not OK
    System.out.println("Number  :   "+number );
}
```

# Final variable

- We can provide value to the final variable either at compile time or run time.

```
public static void main(String[] args) {
    Scanner sc = new Scanner( System.in);
    System.out.print("Number    :   ");
    final int number = sc.nextInt();    //OK
    //number = number + 5;    //Not OK
    System.out.println("Number   :   "+number );
}
```

# Final field

- once initialized, if we don't want to modify state of any field inside any method of the class( including constructor body ) then we should declare field final.

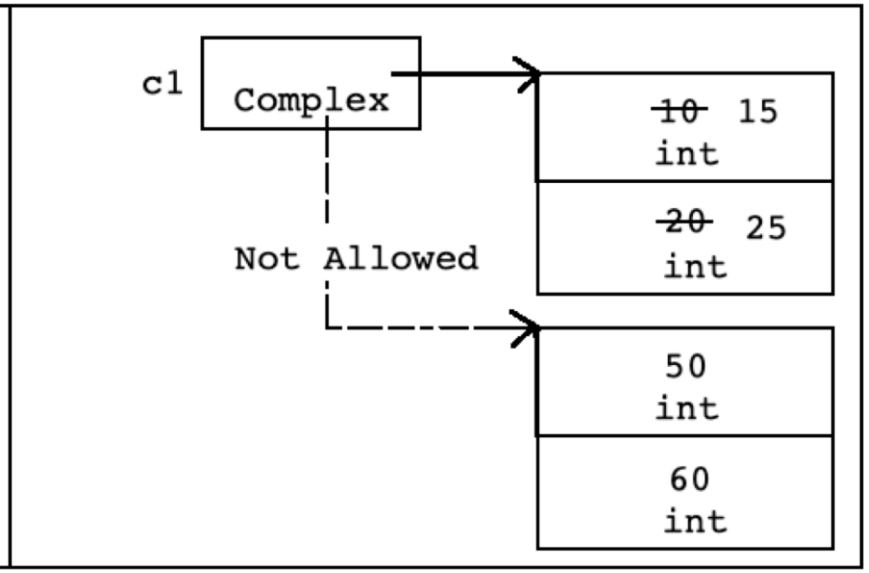
```
class Circle{
    private float area;
    private float radius = 10;
    public static final float PI = 3.142f;
    public void calculateArea( ){
        this.area = PI * this.radius * this.radius;
    }
    public void printRecord( ){
        System.out.println("Area    :   "+this.area);
    }
}
```

- If we want to declare any field final then we should declare it static also.

# Final Reference Variable

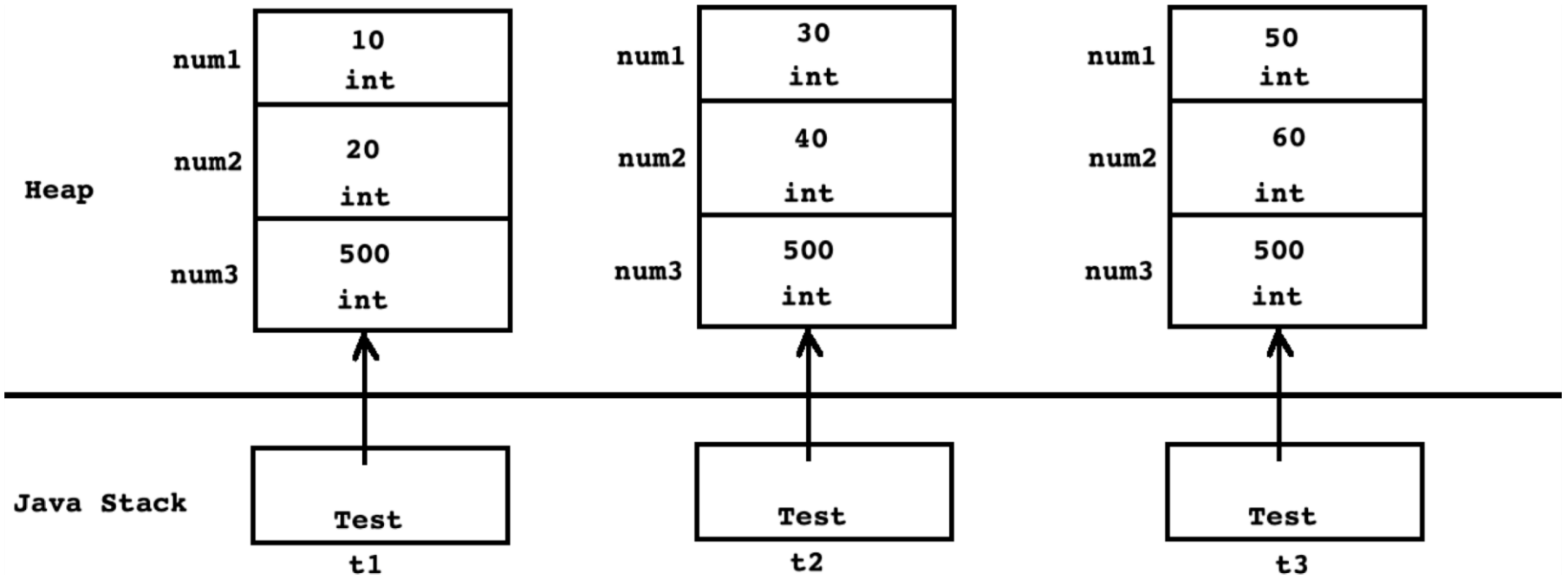
- In Java, we can declare reference final but we can not declare instance final.

```
public static void main(String[] args) {
    final Complex c1 = new Complex( 10, 20 );
    c1.setReal(15);
    c1.setImag(25);
    //c1 = new Complex(50,60);    //Not OK
    c1.printRecord( );      //15, 25
}
```



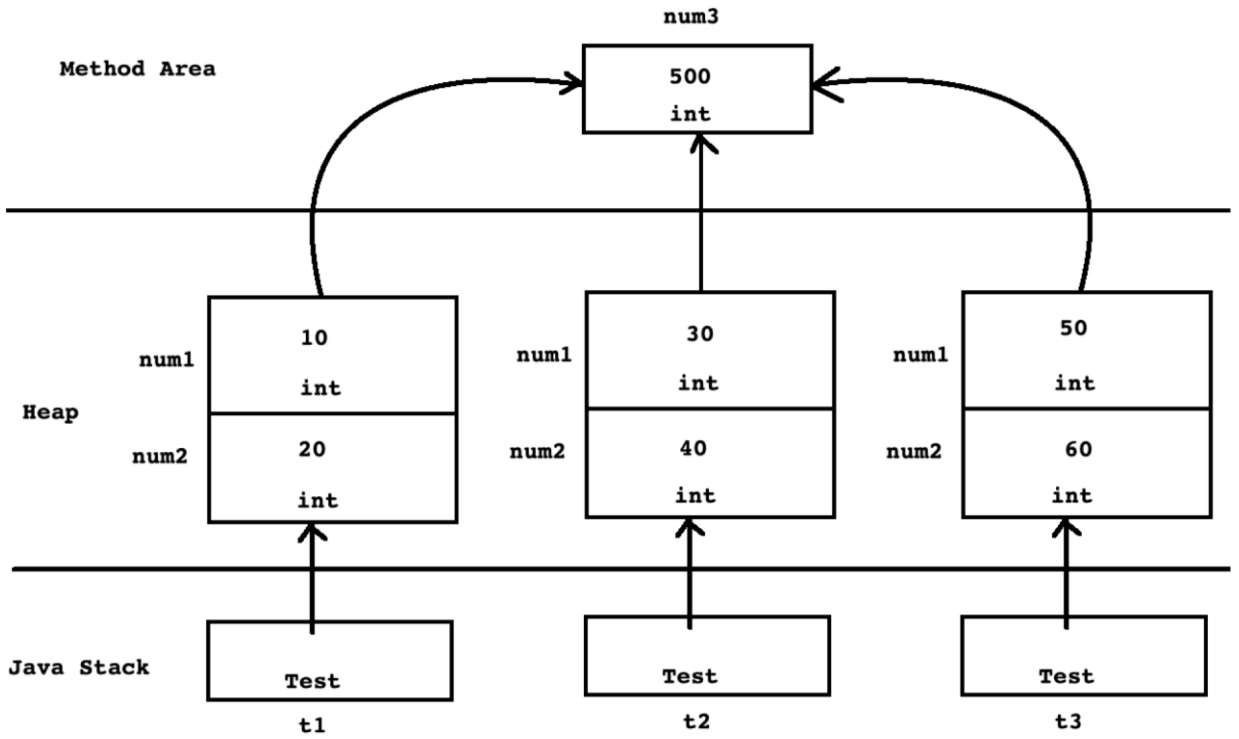

- We can declare method final. It is not allowed to override final method in sub class.
- We can declare class final. It is not allowed to extend final class.

# Static Field



# Static Field

- If we want to share value of any field inside all the instances of same class then we should declare that field static.



# Static Field

- Static field do not get space inside instance rather all the instances of same class share single copy of it.
- Non static Field is also called as instance variable. It gets space once per instance.
- Static Field is also called as class variable. It gets space once per class.
- Static Field gets space once per class during class loading on method area.
- Instance variables are designed to access using object reference.
- Class level variable can be accessed using object reference but it is designed to access using class name and dot operator.

# Static Initialization Block

- A *static initialization block* is a normal block of code enclosed in braces, { }, and preceded by the static keyword. Here is an example:

```
static {
    // whatever code is needed for initialization goes here
}
```

- A class can have any number of static initialization blocks, and they can appear anywhere in the class body.
- The runtime system guarantees that static initialization blocks are called in the order that they appear in the source code.
- There is an alternative to static blocks — you can write a private static method:

# Instance Initializer Block

- Normally, you would put code to initialize an instance variable in a constructor.
- There are two alternatives to using a constructor to initialize instance variables: initializer blocks and final methods.
- Initializer blocks for instance variables look just like static initializer blocks, but without the static keyword:

```
{
    // whatever code is needed for initialization goes here
}
```

- The Java compiler copies initializer blocks into every constructor. Therefore, this approach can be used to share a block of code between multiple constructors.

# Instance Initializer Block

- A *final method* cannot be overridden in a subclass.

```
class Whatever {
    private varType myVar = initializeInstanceVariable();

    protected final varType initializeInstanceVariable() {

        // initialization code goes here
    }
}
```

- This is especially useful if subclasses might want to reuse the initialization method. The method is final because calling non-final methods during instance initialization can cause problems.

# Static Method

- To access non static members of the class, we should define non static method inside class.
- Non static method/instance method is designed to call on instance.
- To access static members of the class, we should define static method inside class.
- static method/class level method is designed to call on class name.
- static method do not get this reference:
    1. If we call, non static method on instance then method get this reference.
    2. Static method is designed to call on class name.
    3. Since static method is not designed to call on instance, it doesn't get this reference.

# Static Method

- this reference is a link/connection between non static field and non static method.
- Since static method do not get this reference, we can not access non static members inside static method directly. In other words, static method can access static members of the class only.
- Using instance, we can use non static members inside static method.

```
class Program{
    public int num1 = 10;
    public static int num2 = 10;
    public static void main(String[] args) {
        //System.out.println("Num1    :   "+num1);    //Not OK
        Program p = new Program( );
        System.out.println("Num1    :   "+p.num1);    //OK
        System.out.println("Num2    :   "+num2);
    }
}
```

# Static Method

- Inside method, If we are going to use this reference then method should be non static otherwise it should be static.

```
class Math{
    public static int power( int base, int index ){
        int result = 1;
        for( int count = 1; count <= index; ++ count ){
            result = result * base;
        }
        return result;
    }
}
```

```
class Program{
    public static void main(String[] args) {
        int result = Math.power(10, 2);
        System.out.println("Result  :   "+result);
    }
}
```

# Static Import

- If static members belonging to the same class then use of type name and dot operator is optional.

```
package p1;
public class Program{
    private static int number = 10;
    public static void main(String[] args) {
        System.out.println("Number  :   "+Program.number);  //OK    :   10
        System.out.println("Number  :   "+number);  //OK    :   10
    }
}
```

# Static Import

- If static members belonging to the different class then use of type name and dot operator is mandatory.
- PI and pow are static members of java.lang.Math class. To use Math class import statement is not required.
- Consider Following code:

```
package p1;
public class Program{
    public static void main(String[] args) {
        float radius = 10.5f;
        float area = ( float )( Math.PI * Math.pow( radius, 2 );
        System.out.println( "Area    :    "+area );
    }
}
```

# Static Import

- There are situations where you need frequent access to static final fields (constants) and static methods from one or two classes. Prefixing the name of these classes over and over can result in cluttered code. The static import statement gives you a way to import the constants and static methods that you want to use so that you do not need to prefix the name of their class.
- Consider code:

```
package p1;
import static java.lang.System.out;
import static java.lang.Math.*;
public class Program{
    public static void main(String[] args) {
        float radius = 10.5f;
        float area = ( float )( PI * pow( radius, 2 );
        out.println( "Area   :   "+area );
    }
}
```


**Thank you.**

**akshita.Chanchlani@sunbeaminfo.com**